PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack

Daimeng Wang
University of California, Riverside
Riverside, CA
dwang030@cs.ucr.edu

Zhiyun Qian
University of California, Riverside
Riverside, CA
zhiyunq@cs.ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
Riverside, CA
nael@cs.ucr.edu

Srikanth V Krishnamurthy
University of California, Riverside
Riverside, CA
krish@cs.ucr.edu

ABSTRACT

CPU memory prefetchers can substantially interfere with prime and probe cache side-channel attacks, especially on in-order CPUs which use aggressive prefetching. This interference is not accounted for in previous attacks. In this paper, we propose PAPP, a Prefetcher-Aware Prime Probe attack that can operate even in the presence of aggressive prefetchers. Specifically, we reverse engineer the prefetcher and replacement policy on several CPUs and use these insights to design a prime and probe attack that minimizes the impact of the prefetcher. We evaluate PAPP using Cache Side-channel Vulnerability (CSV) metric and demonstrate the substantial improvements in the quality of the channel under different conditions.

KEYWORDS

CPU cache, Prime and probe, Prefetching

ACM Reference Format:
Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. 2019. PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317877

1 INTRODUCTION

CPU cache side-channels can be exploited to extract sensitive information [15, 20]. These attacks can be launched by any userspace process and are able to bypass cross-process and even cross-VM boundaries. The most general of these attacks is Prime and Probe which has been widely used to extract secret keys from crypto algorithms including AES [7, 15, 17] and El-Gamal [12]. To successfully implement a prime and probe attack, researchers put considerable efforts into understanding and reverse engineering CPU features such as cache indexing [5, 13] and replacement policy [1].

One aspect of side-channel attack that has not been well studied is the impact of the data prefetcher, which speculatively fetches unaccessed cache lines to improve cache hit rate. The prefetcher adds noise to the side-channel information in two ways: first, the victim signal has some spurious accesses that are from the prefetcher rather than the application. Moreover, the attacker's prime and probe access patterns are limited since it also generates unneeded memory accesses from the prefetcher. The effects described above can substantially interfere with prime and probe attacks. One commonly adopted methodology is to utilize a linked-list setup of the eviction set [17] to suppress prefetching. By adopting this approach, an attacker can suppress the prefetcher to the next-line prefetcher only and thus enable prime and probe of every other cache set. Still, this approach leaves the attacker missing half of the cache sets, lowering the quality of the leaked signal. To make things worse, on CPUs with more aggressive prefetching, the attacker might not even be able to prime and probe every other set, making the attack way less effective. In particular, in-order processors such as the Intel Atom, which is often used in embedded systems, have very aggressive prefetchers since cache misses cause substantial performance losses without the support of out-of-order execution.

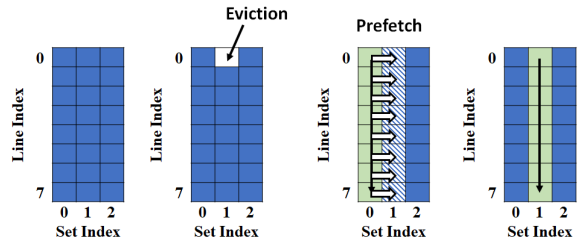
In this paper, we develop a new prime and probe attack which we call Prefetcher-Aware Prime and Probe (PAPP). PAPP reverse-engineers the replacement policy and the prefetching behavior and generate a prime and probe pattern in an automated fashion. PAPP mitigates the impact of the prefetcher on prime and probe attacks, substantially improving their effectiveness (especially in the presence of aggressive prefetchers). Our experiments show that PAPP can almost completely circumvent the effect of cache prefetching on in-order CPUs with aggressive prefetching policy, substantially improving the quality of prime and probe attack.

Our main contributions are:

- We perform a systematic study on the impact of prefetching on prime and probe attacks. We demonstrate limitations of the existing implementation of prime and probe attack.
- We present a novel prime and probe strategy aiming to address the effect of prefetching. We combine the knowledge of replacement policy and cache prefetcher to effectively circumvent the effect of cache prefetching. We automate the generation of prime and probe strategy and open-source our implementation at [9].
- We evaluated PAPP on real-world system using cache side-channel vulnerability (CSV) metric. We show that our approach doubles the information leakage comparing to traditional prime and probe implementations. We also discuss the effect of the prefetcher on CSV metric.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317877



(a) Initial state (b) Victim's eviction (c) Signal erased (d) Cache hit

Figure 1: Prefetcher's effect on traditional prime and probe

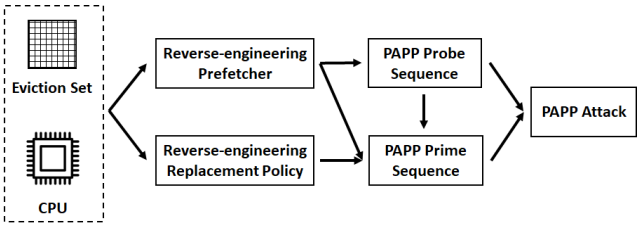


Figure 2: PAPP Attack Workflow

2 BACKGROUND AND MOTIVATION

The high cost of memory accesses is one of the fundamental bottlenecks limiting processor performance. Processors use caches (often *Set-associative*) to store recently accessed memory in order to compensate this cost. Moreover, modern CPUs exploit predictable program access patterns using prefetchers that preload memory that is likely to be accessed in the future [18]. Prefetchers can be quite aggressive, especially in in-order processors which are often used in embedded applications: in such processors, cache misses cannot be compensated for using out-of-order execution. For example, Intel CPUs implement a streaming prefetcher which could prefetch up to 20 cache lines ahead [6].

Because CPU caches are shared among multiple programs, they become targets of side-channel attacks [15]. Prime and probe is one of the most general attack strategies because it does not require shared memory pages with the victim. At high level, the attacker starts with completely filling (*prime*) the cache sets she wish to monitor using a carefully chosen *eviction set*. When the victim generates memory references, its accesses replace some of the cache lines in the eviction set filled by the attacker. The attacker can then access the eviction set again (*probe*); whenever an access results in a cache miss, she can infer that the victim has accessed that cache set resulting in her data being replaced.

The most commonly used approach to implement a prime and probe attack is to sequentially access all cache lines in the same cache set from the eviction set, timing the total access time [12, 15]. This way, priming and probing are built into one access pattern, making it efficient to monitor a single cache set.

Unfortunately, this approach does not work well with multiple cache sets in the presence of a prefetcher. Figure 1 shows an example of a prime and probe attack on 8-way set-associative cache. First, the attacker primes all cache lines in sets 0,1,2 and waits for victim's activity (a). Next, victim's memory activity evicts cache set 1 (b). In (c), the attacker starts to prime and probe cache set 0, getting all cache hits due to the lack of victim's activity on this set. However, during this process, the prefetcher loads all cache lines from cache set 1 back, completely erasing the signal created by the victim (d).

This issue cannot be solved by simply going backward in prime and probe order. On many modern CPUs, the prefetcher can also detect whether an attacker accessing memory in forward order or backward order, changing prefetching tactics accordingly. Tromer et. al. [17] attempted to set up a random-order linked-list structure and utilize a pointer-chasing technique when accessing eviction set memories, suppressing the stream prefetcher from aggressively loading too many cache sets because of the unpredictable access pattern. However, we find that this approach still cannot completely get around the next-line prefetching (a standard prefetcher that always brings in the next cache line), especially on in-order CPUs with more aggressive prefetching. As a result, attackers often compensate for this prefetching issue by either skipping every other cache set and/or repeatedly conducting experiments and testing different cache sets each time [23]. In either case, the performance and precision of prime and probe is severely impacted, potentially making it unusable for CPUs with aggressive prefetchers.

3 PAPP DESIGN AND IMPLEMENTATION

We propose Prefetcher-Aware Prime and Probe (PAPP) attack, a prime and probe attack that overcomes the negative impacts of prefetching. Figure 2 demonstrates a high-level workflow of the attack. Similar to other prime and probe attacks, the attacker first creates an **eviction set**. Utilizing this eviction set, PAPP first conducts **prefetcher reverse engineering** and **replacement policy reverse engineering**. Using the obtained profiles of the prefetcher and replacement policy, PAPP then constructs a **probe sequence** and subsequently a **prime sequence**. PAPP then combines both sequences into a prime and probe attack that circumvents the interference of the prefetcher.

Specifically, PAPP leverages two new ideas: (1) It first reverse-engineers the replacement policy to make the probe sequence possible using only one access to each set (instead of having to access all the cache lines in each set). As a result, fewer accesses are generated, leading to less prefetching activities; and (2) It uses probe patterns that avoid the impact of the prefetcher. Together, the techniques allow for near perfect probing of the cache, even on in-order CPUs with aggressive prefetching. Although we omit the details due to space, we are able to automate the construction of these sequences giving our profiling, potentially enabling the attack to be adaptable with little effort to other CPUs. Detailed implementation of algorithms used by PAPP can be found at [9].

To simplify explanations, and without loss of generality, we assume that the attacker targets a single memory page. We design and implement PAPP attack on Intel Atom Z3560 and Z3580 and use the L2 cache of Z3580 for demonstration. Intel Atom is an in-order processor with a unified 16-way set-associative L2 cache. It has a cache line size of 64 bytes, which means it would require 64 cache sets to cover a 4KB memory page. We selected the Atom, as an in-order processor representative of what is used in embedded systems, because such processors are known to use aggressive

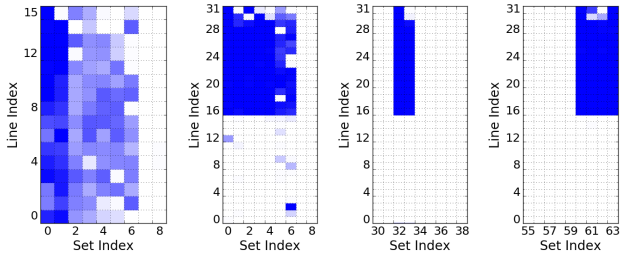


Figure 3: Occupancy on L2 cache of Atom Z3580. Darker cells means higher chance of cache line being in cache. Each cache line tested 100 times.

prefetchers. All results are from real experiments conducted on an Android phone using Atom Z3580 CPU.

3.1 Reverse Engineering the Prefetcher and Replacement Policy

Eviction Set: We construct the eviction set ES with m sets and n lines per set similar to other prime and probe attacks (e.g., [7]). Let a be the associativity of the CPU cache level being targeted, i.e. each cache set is composed of a cache lines. Therefore, in order to fully occupy the cache, we need $n \geq a$. We partition the eviction set ES in to two sections: **occupation section** and **warmup section** (which is a new improvement we introduce here). The occupation section is composed of $m \times a$ cache lines, designed to occupy the cache after priming. This section is essential to all prime and probe attacks. On CPUs with strict least-recently-used (LRU) replacement policy, the warmup section is not needed. However, most modern CPUs do not have a strict LRU policy [1], hence the necessity of the warmup section which is used to make the occupancy and replacement state of the cache set more predictable as we demonstrate later in the paper.

Reverse Engineering Prefetcher: We reverse engineer the CPU memory prefetcher and use the results later in Section 3.2 to construct PAPP prime and probe sequence. Prefetcher reverse engineering aims to answer the following two questions:

(1) What cache lines will be occupying the CPU cache after accessing a sequence of memories in the eviction set?
(2) What cache lines will be first replaced upon victim's access to the same cache set?

We perform prefetcher reverse-engineering using two steps: 1) Access a sequence of cache lines in the eviction set ES and 2) Access an arbitrary cache line in ES and check whether the this line is cached (indicating it was loaded or prefetched). We adopt the pointer-chasing technique used in previous literature [17] during the accesses suppress the prefetcher.

The prefetcher behavior is complex and can vary based on the access pattern and the availability of memory bandwidth. To provide a basic characterization, we conduct prefetcher reverse-engineering with respect to accesses to a single set, i.e. what cache lines will be in the cache after we fully prime a single set s in by accessing all cache lines mapped to s in the eviction set. The reverse-engineering result on Intel Atom Z3580 processor is shown in Figure 3.

Figure 3(a) demonstrate the cache occupancy heat map of the eviction set after priming set $s = 0$ in the eviction set without any warmup section (i.e. $n = a = 16$). We notice that Atom does not have a naive LRU replacement policy, as accessing 16 cache lines in the eviction set does not guarantee full cache occupancy of the accessed lines. As a result, we determine that a warmup section is necessary for reliable cache priming.

Figures 3(b), 3(c), 3(d) shows the cache occupancy heat map after priming set 0, 32 and 63 respectively with 16 extra lines as warmup (i.e. $n = a + 16 = 32$). First, we see that using 16 extra lines reliably ensures the cache occupancy of the occupancy section of the eviction set (lines 16-31). Additionally, we found that the prefetching behavior differs for different cache sets. The prefetcher is more aggressive at the beginning of the page (cache set 0) than at the middle of the page (cache set 32). And towards the end of the page (cache set 63) the prefetcher will prefetch previous cache set instead of next cache set apparently to avoid prefetching into a potentially unmapped or uncached page.

Based on the analysis above, we conclude that although traditional prime and probe will be able to monitor every other set at the middle of the memory page, it's impossible to do so at the beginning and end of the page. The effectiveness of traditional prime and probe is substantially impacted by aggressive prefetching.

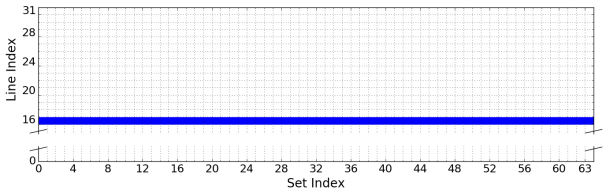
Reverse Engineering Replacement policy: We reverse engineer the replacement policy which is needed to construct the prime and probe sequences. If we can reliably set up which cache line will be replaced if a set is accessed by the victim, we can probe only this line to determine if there is a victim access, substantially reducing the number of memory accesses and prefetcher noise.

Similar to prefetcher profiling conducted in Figure 3, we conduct replacement profiling by first fully priming a cache set s. Afterwards, for each cache line in s, we compare the miss rate of this cache line with/without a single victim memory access in set s as an indicator of how often a cache line in the eviction set will be replaced after a single victim memory access at the same cache set. Figure 4 shows our profiling result on Intel Atom Z3580. According to Figure 4(a), when we have a 16-cache-line warmup section, the 16th least recently used cache line reliably becomes the next victim. However, as we can see in Figure 4(b), with a warmup section size of 8, the replacement status becomes much less predictable.

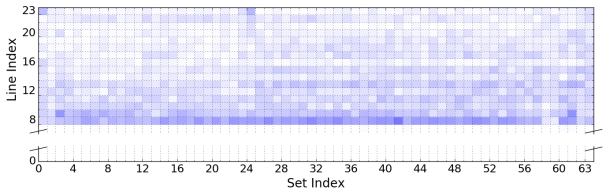
3.2 PAPP Prime and probe Sequence

With the knowledge of the prefetching and replacement policies, PAPP crafts a prefetcher aware prime and probe sequence.

Probe Sequence: To avoid prefetcher effects, accesses in the probe sequence should not prefetch memory from the rest of the sequence; otherwise prefetched data overwrites any victim data. Moreover, accessing probe sequence should not prefetch new data into the cache; otherwise prefetched data evicts attacker's eviction set data. We have automated the construction of probe sequence using the prefetcher reverse-engineering result we discussed in Section 3.1. We start with an empty sequence $seq_{probe} = ()$. For each cache set s in the eviction set ES, we try to find a cache line indexed to s



(a) $n = a + 16$



(b) $n = a + 8$

Figure 4: Replacement profiling on L2 cache of Atom Z3580. Darker means higher chance of line being replaced first. Each cache line tested 100 times.

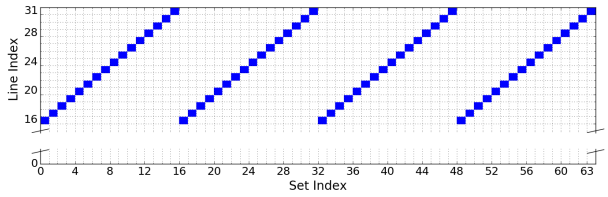


Figure 5: Sample probe sequence for L2 cache of Intel Atom.

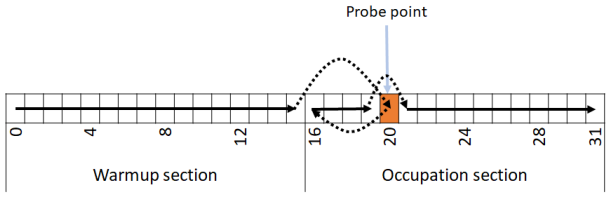


Figure 6: Sample prime sequence (solid line) for L2 cache of Intel Atom which sets line 20 to be replaced next.

in the occupation section of *ES* (thus prevents self-evicting) such that accessing seq_{probe} does not prefetch the chosen cache line. Upon finding such cache line, we append it to seq_{probe} and move on to the next set until all sets are covered. There can be multiple possible probe sequences that satisfy our requirements. Figure 5 illustrates one possible generated probe sequence for L2 cache of Intel Atom Z3580.

Prime Sequence: For each cache set s, we seek a memory access sequence of cache lines in *ES* that can (1) occupy the cache with the occupancy section in *ES* and (2) set the chosen cache line in the probe sequence to be the next one being replaced. We automate this process utilizing the reverse-engineering result of the prefetcher and replacement policy. Figure 6 shows a prime sequence of one set generated on Intel Atom Z3580. The sequence first access all cache lines in the warmup section. Next, it accesses line 20 and finally rest of cache lines in occupation section. According to our reverse-engineering result in Figure 4(a), this will reliably set line 20 as the next one being replaced.

Generating prime sequence will be problematic when there is forward prefetching and backward prefetching at the same time. In the case of Atom Z3580, priming cache set 62 will prefetch all cache lines in set 61 while priming cache set 61 will prefetch set 62. Therefore whichever set got primed first will have its replacement status wiped when priming the other set. Fortunately, for Atom Z3580, such backward prefetching only exists when priming cache set 62 and 63. In practice, the attacker can omit these two cache sets in order to ensure other cache sets are monitored effectively.

4 EVALUATION

We implement both PAPP and traditional prime and probe attacks in C and perform the attack on Intel Atom Z3580 CPU running Android OS. For traditional prime and probe attack, we use a standard implementation following previous attacks [4, 7, 23] which prime and probe every other cache set and uses pointer chasing to suppress the prefetcher. In contrast, PAPP is able to probe all sets with the exception of sets 62 and 63 as discussed earlier. We use a benchmark victim program and test the ability of prime and probe attack towards inferring victim's activity. We do not apply any prefetcher suppression techniques to the victim. We consider three victim process access patterns:

(1) Accessing one cache line: in this pattern, the victim's behavior corresponds to LLC-based AES attacks introduced in [4, 7, 23]. In these attacks, the attacker exploits the Linux complete fair scheduler (CFS) to interrupt the victim's execution. As a result, the victim can only access a single AES table entry between two attacker prime and probe rounds.
(2) Accessing six consecutive cache lines: corresponds to attacks on El-Gamal cipher [12], where each table entry spans 6 cache lines. CFS exploit is optional since the crypto computation is much slower than AES.
(3) Accessing a various number of random cache lines: this case provides a tunable more general access pattern.

To measure the effectiveness of the attack, we use the cache side-channel vulnerability (CSV) [21] metric. CSV computes Pearson correlation coefficient between the victim's cache activity (oracle) and attacker's measurement. The higher the CSV, the stronger the correlation between victim's activity and attacker's measurement, indicating better attack effectiveness.

4.1 Modification to CSV Metric

CSV does not account for CPU cache prefetching and has only been used in experiments where the prefetcher is disabled. The original work [21] assumed that the prefetcher can be either disabled or fully suppressed. As a result, it only considers the victim accesses from the application perspective rather than their footprint in the cache as observed by the attacker. In reality, however, the prefetcher affects victim's memory footprint and in turn affect attacker's observations.

Figure 7 demonstrates one attack scenario. In this example, the victim accesses one cache set while an attacker (traditional or PAPP)

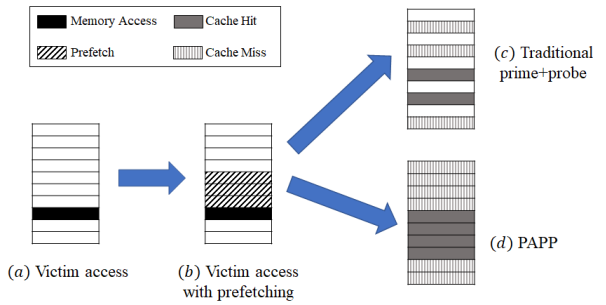
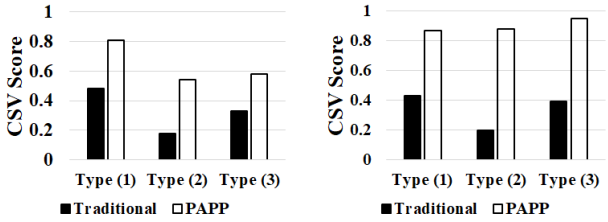


Figure 7: Victim's prefetching



(a) Next-line prefetch prediction. (b) Full prefetch prediction.

Figure 8: CSV score with prefetch prediction.

is trying to infer victim's access. To measure the effectiveness of the attacker, CSV computes the correlation between attacker's observation (Figure 7(c), Figure 7(d)) and victim's access (Figure 7(a)). With the presence of prefetcher, however, the attacker does not directly monitor victim's memory access. Instead, attacker can only monitor a combination of victim's memory access and victim's prefetched memory access (Figure 7(b)). Therefore, computing the correlation between (a),(c) and (a),(d) does not accurately reflect the success in recovering the cache state, as some of the attacker's observations can only correlate to the victim's prefetching behavior.

To address this issue, for the victim access pattern we include the prefetching behavior based on a model of the prefetcher. A simple model is to assume that the prefetcher will only prefetch next cache line for the victim. A more sophisticated model can use the prefetcher profile (as we carried out in the previous section) to more accurately predict victim's prefetching behavior. Note that we only modify the computation of the CSV metric, not to the operation of PAPP or traditional prime and probe experiments.

4.2 Comparison to Traditional Prime and Probe

Figure 8 shows the CSV score with the traditional attack and PAPP. We found that for type (1) victim, the prefetcher indeed only prefetches the next cache line except at the beginning and end of a page. PAPP substantially outperforms traditional prime and probe across all cases: for example, for type(1) workload, PAPP achieve a CSV of 0.81 using the modified next line CSV metric (Figure 8(a)) and even higher with the full prefetch prediction (Figure 8(b)), while traditional prime and probe scores only 0.48, demonstrating that PAPP is a much higher quality attack. Since traditional prime and probe can only probe every other cache set, it cannot capture access to sets not being probed, resulting in a lower correlation.

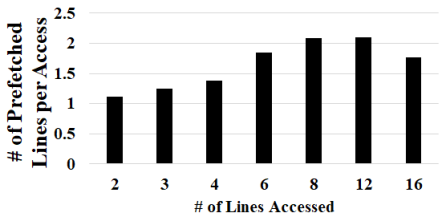


Figure 9: Prefetched cache lines for type (3) victim.

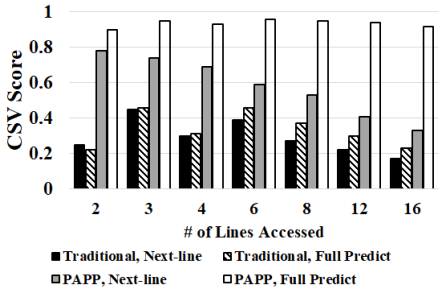


Figure 10: CSV for type (3) victim

For type (2) and type (3) victims, we found that the prefetcher is more aggressive when multiple closely-located cache lines are accessed. In type (2) victim, we found that the prefetcher is constantly prefetching 10-11 cache lines. In type (3) victim, the aggressiveness of the prefetcher depends on the memory access pattern of the victim. As a result, a naive next-line prediction can no longer reflect the cache activity of the victim program. This can be compensated by having a better profile of victim's prefetching behavior, as shown in Figure 8(b). We see that with the correct prefetcher profiling, the CSV for type (2) victim rises to 0.2 (traditional) and 0.88 (PAPP), while the CSV for type (3) victim rises to 0.39 (traditional) and 0.95 (PAPP). We notice that type (2) victim has a low CSV under traditional prime and probe. This is because the prefetcher is more aggressive for type (2) victim, hence prefetching cache lines in more sets that traditional prime and probe cannot monitor.

Figure 9 further demonstrates the effect of the prefetcher when a type (3) victim accesses various numbers of cache sets. On average, the prefetcher prefetches no more than 2.1 additional lines per victim's memory access. Figure 10 shows the CSV score for type (3) victim with different number of accesses. We notice that as the intensity of the victim's activity increases, next-line prediction approach loses its effectiveness. A refined "full prediction" is necessary when victim accesses more memory during one round of prime and probe. As we mentioned in Section 4.1, this observation only refers to the computation of the CSV metric. The operations of PAPP or traditional prime and probe remain unmodified.

4.3 Discussion

PAPP exploits the replacement policy and attempts to set the cache replacement to a more predictable status. Through our experiment with Intel Atom processors, we show that this can be achieved on processors without naive LRU replacement policy. On ARM

CPUs, however, we are unable to achieve the same since ARM's replacement policy is much less predictable as demonstrated in previous research. [10]

We also experimented on some out-of-order Intel CPU architectures including Xeon, Haswell, Sandybridge and Skylake. PAPP can successfully generate a prime and probe strategy on these CPUs but their prefetchers behaves very differently from Atom. On Xeon, we found that pointer chasing technique can effectively disable next-line prefetcher. On Haswell, Sandybridge and Skylake, we found that priming a cache set with odd index will prefetch the next line while priming a cache set with even index will prefetch the previous line. As we discussed in Section 3.2, the generated prime and probe strategy cannot monitor adjacent cache sets and achieve higher coverage than traditional prime and probe on these CPUs One conclusion of this is that PAPP is most important for in-order processors which have more aggressive prefetching.

Besides coverage, PAPP also has an advantage of manipulating the replacement policy and separating probe sequence with prime sequence. Specifically, when probing results in a cache hit, an attacker can simply skip this set during priming, improving the throughput of prime and probe drastically. This benefit applies to both in-order CPUs with aggressive prefetchers and out-of-order CPUs with less aggressive prefetchers.

We show that with the prefetcher-aware approach of PAPP, we can monitor more cache sets than traditional prime and probe attacks. We believe this makes PAPP applicable on a wider variety of attacks, especially in scenarios where attacker can only obtain a limited number of observations. (e.g. [19]) We plan to implement new cache side-channel attacks using PAPP in the future.

5 RELATED WORK

There has been an abundance of existing work on prime and probe CPU cache side-channel attacks. The Advanced Encryption Standard (AES) is the first to fall victim to prime and probe attack [7, 15, 17] where attackers were able to recover victim's memory accesses of AES lookup table, inferring the secret key. Prime and probe attack is also used to break other mechanisms such as El-Gamal [12]. Zhang et al. [22, 23] show that prime and probe attacks can even cross VM boundaries and perform cross-tenant attacks on PaaS (Platform as a service) clouds. Moreover, prime and probe is shown to work not only on Intel CPUs but also other environments such as ARM [10] and browsers [14].

There have been a few studies on prefetcher's effect on cache side-channel attacks. Tromer et al. [17] is the first to acknowledge CPU prefetching's interference on prime and probe attack and propose a linked-list structure of eviction set and pointer-chasing technique to suppress the prefetcher. This technique is widely adopted in almost all known prime and probe implementations. Fuchs et al. [3] demonstrate that it is possible to defend prime and probe attacks by applying disruptive prefetching techniques to obfuscate victim's memory footprint. Unfortunately, to our knowledge, this technique is not implemented in any CPUs.

Recently, researchers are paying more attention to the implications of CPU optimizations (e.g. prefetcher, branch predictor, etc.) on side-channel attacks. Shin et al. [16] show that CPU cache stride prefetching introduces a side-channel that can be exploited against ECDH algorithm in OpenSSL. Other CPU optimizations such as branch predictor (e.g. [2]) and speculative execution (e.g. [8, 11] have also been exploited for side-channel attacks.

6 CONCLUSION

In this paper, we propose PAPP: a prefetcher-aware prime and probe cache side-channel attack. PAPP performs systematic reverse-engineering of CPU cache prefetcher and replacement policy. We show that PAPP is able to construct prime and probe strategy that are resistant to the interference of aggressive prefetchers on in-order CPUs. We evaluated PAPP on real-world system using cache side-channel vulnerability (CSV) metric and demonstrates that PAPP doubles the information leakage comparing to traditional prime and probe implementations. We hope that in the future, PAPP can be used in new attacks and applications to provide efficient cache monitoring.

REFERENCES

[1] A. Abel and J. Reineke. 2014. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *Proc. ISPASS '14.*

[2] D. Evtyushkin, P. Ponomarev, and N. Abu-Ghazaleh. 2016. Jump-over-ASLR: Attacking the Branch Predictor to Bypass ASLR. In *Proc. Micro '16.*

[3] A. Fuchs and R. B. Lee. 2015. Disruptive Prefetching: Impact on Side-channel Attacks and Cache Designs. In *Proc. SYSTOR '15.*

[4] D. Gullasch, E. Bangerter, and S. Krenn. [n. d.]. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proc. IEEE SP '11.*

[5] R. Hund, C. Willems, and T. Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proc. IEEE SP '13.*

[6] Intel. 2016. Intel 64 and IA-32 Architectures Optimization Reference Manual). (2016). https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[7] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *Proc. DAC '16.*

[8] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Proc. IEEE SP '19.*

[9] UCR Security Lab. 2019. PAPP Repo. (2019). https://github.com/seclab-ucr/PAPP

[10] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security '16.*

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and . Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. USENIX Security '18.*

[12] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B Lee. 2015. Last-level cache side-channel attacks are practical. In *Proc. IEEE SP '15.*

[13] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. 2015. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Proc. RAID '15.*

[14] Y. Oren, V. P Kemerlis, S. Sethumadhavan, and A. D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proc. SIGSAC '15.*

[15] D. A. Osvik, A. Shamir, and E. Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval (Ed.).

[16] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur. 2018. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *Proc. SIGSAC '18.*

[17] E. Tromer, D. A. Osvik, and A. Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23 (01 Jan 2010).

[18] S. P Vanderwiel and D. J Lilja. 2000. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)* 32, 2 (2000).

[19] D. Wang, A. Neupane, Z. Qian, N. Abu-Ghazaleh, S. V Krishnamurthy, E. JM Colbert, and P. Yu. 2019. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In *Proc. NDSS '19.*

[20] Y. Yarom and K. Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. USENIX Security '14.*

[21] T. Zhang, F. Liu, S. Chen, and R. Lee. 2013. Side Channel Vulnerability Metrics: The Promise and the Pitfalls. In *Proc. HASP '13.*

[22] Y. Zhang, A. Juels, K Reiter, M., and T. Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proc. SIGSAC '14.*

[23] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proc. CCS '12.*